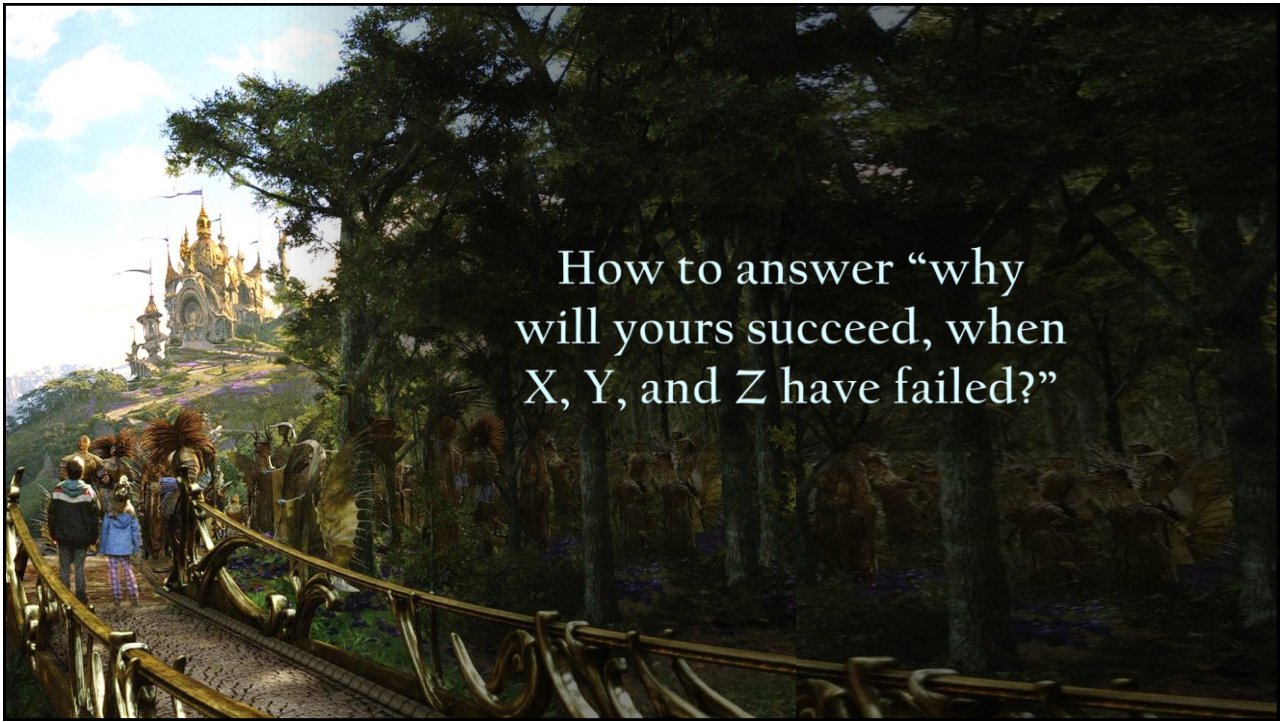


1



2

~~Out of scope~~
“Necessary but not sufficient”

Table stakes:

- funding
- team
- quality
- rollout
- marketing
- ...

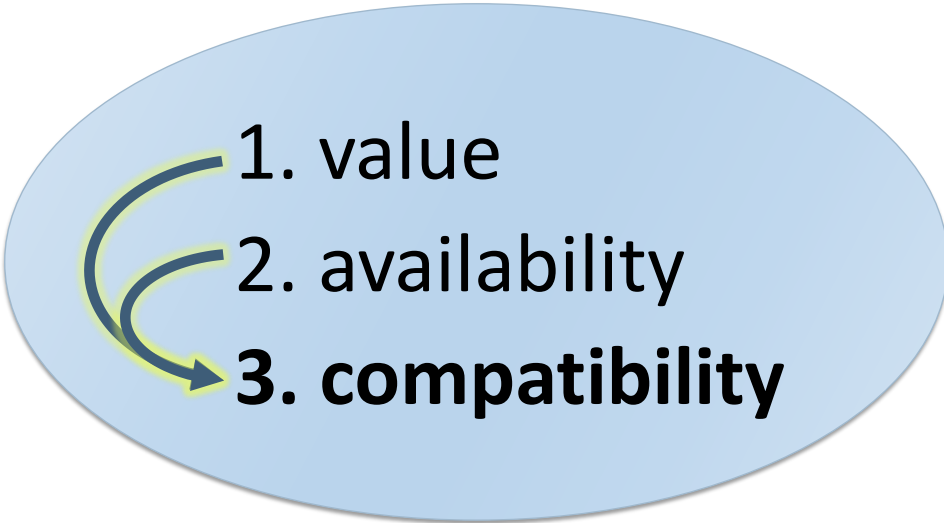
In scope
Strategy differentiators

What did SuccessfulThing
design for so that it succeeded
when X, Y, and Z didn’t?

1. value
2. availability
- 3. compatibility**

4

4



5

5

Definition: “NewThing”

Can be a competing new product

Examples: D, Go, Swift, Rust

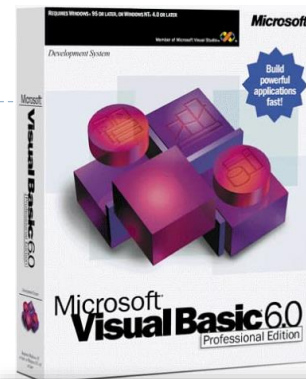
Can be “vNext” of the same-brand product

Same thing: For an established product,

vNext’s biggest initial competitor is vPrev

Examples: C99, C++20, VB .NET, Python 3

“Same brand/name” doesn’t mean users will accept as same and upgrade → **compatibility**



Visual Basic

From Wikipedia, the free encyclopedia

This article is about the Visual Basic language shipping with Microsoft Visual Studio 6.0 or earlier. For the Visual Basic language shipping with Microsoft Visual Studio .NET or later, see [Visual Basic .NET](#).

6

6

programming languages

1969: Jean Sammet listed 120 (very selective)

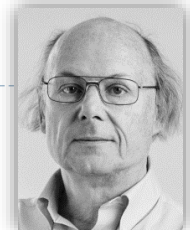
c. 2000: Bjarne Stroustrup estimated **10,000**

Sampled several years, all available sources: ~200/year

Counted named dialects, macro-assemblers, academic languages used only by small research groups, specialized languages (e.g., build, process control), and corporate languages (internal, external)

~**100s of “survivors”** maintained and used for more than a small group

~**10s of “popular”** languages achieved widespread + long-lived use



Invitation for work
Stroustrup would welcome someone doing an up-to-date validated count

7

7



8

1. NewThing's **value**

Distinct articulable new value...

Clear and explainable in a 30-sec elevator pitch.



Adapted from imgs.xkcd.com/comics/freedom.png

9

9

1. NewThing's **value**

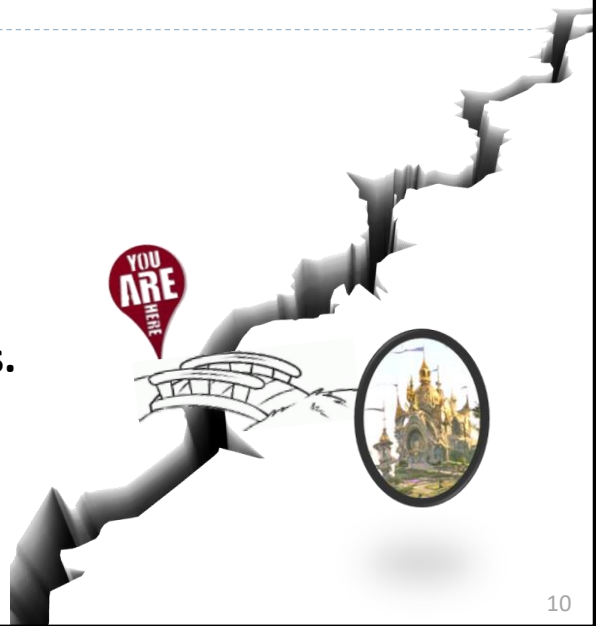
Distinct articulable new value...

Clear and explainable in a 30-sec elevator pitch.

... that solves **known pain points**.

Connection: To OldThing.

Orientation: "We aim to take you from here across this gap."



10

10

Example: JavaScript

Concise elevator pitch:

Well-known pain points with JavaScript



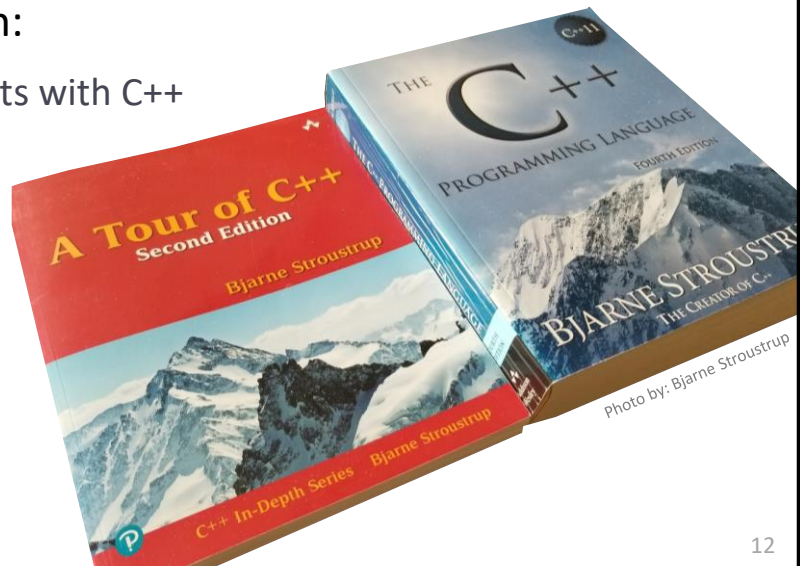
11

11

Example: C++

Concise elevator pitch:

Well-known pain points with C++



12

12

A word about “value”

Classic “shiny NewThing” value mistakes:

1. Underestimating value of OldThing

Examples: “Legacy, old, complex”

Reality: “Proven, working, robust” (with flaws)

2. Underestimating cost of parity with OldThing

Examples: Features, performance, stability

Reality: “90% done, now let’s do the other 90%”

1+2. Underestimating adoption cost

Asking both: Let go old value + adopt new value



13

Counterexample: Visual C++ 6 (1998)

1998: VC++ 6.0 released

2008: Lots of customers still using v6

IDE added many features, but slower & cumbersome

Usability regression is a “UX” incompatibility

Customers complained they could write and debug an MFC program better and faster in v6

2010: VC++ 2010 slogan “10 is the new 6”

Contributing factors: **Binary and source incompatibilities**

Both happened simultaneously \Rightarrow opposite of virtuous cycle



14

14

Counterexample: Simple/terse syntax

“Syntax” is a means, not an end

Not a primary value on its own

It can be part of a means to achieve a valid value goal, such as regularity for learning, or context free parsing for toolability

Easy to overdo when treated as a goal

“Beware of the Turing tar-pit in which everything is possible but nothing of interest is easy.” – Alan Perlis

Even asm doesn’t stay simple

Code	Pseudo code
>>	Move forward 2 to cell2
[-]	Set cell2 to 0
<<	Move back 2 to cell0
[While cell0 is not 0
-	Subtract 1 from cell0
>>	Move to cell2
+	Add 1 to cell2
<<	Move back to cell0
]	End while

Source: esolangs.org \rightarrow search “move to cell2”

Equivalent, without whitespace

>>[-]<<[->+<<]

15

15



1. VALUE

16



2. AVAILABILITY

17

2. NewThing **availability** by design

Requirement: Availability essentially anywhere OldThing is already used + easy to add to my project.

Grail: “Any OldThing project in any environment/platform can easily add NewThing.”

Availability requires **strategic up-front design**.

Must be explicit design goal from the start, but can grow into it.

Example: iPhone designed to be able to work on all existing networks, even if initially targeting only one carrier per region. (Note: Related to **compatibility**...)



18

18

2. NewThing **availability** by design

C: *Designed* to work on a wide range of hardware – “they said [portable performance] couldn’t be done, and he did it.”

Cfront: First C++ compiler *compiled to C*, usable “anywhere C is” – including the C optimizer (perf) and system linker (compat).

CPython: Reference compiler *written in C*, usable “anywhere C is.”

TypeScript: Every TypeScript program *compiles to JavaScript*, runs on any JavaScript runtime.

Swift: Available to *every Xcode developer on every platform* that Objective-C supported (and now more).

Roslyn vNext C# compiler: Available to *every Visual Studio developer on every platform* Visual C# supported.



Cfront

CPython



roslyn

19

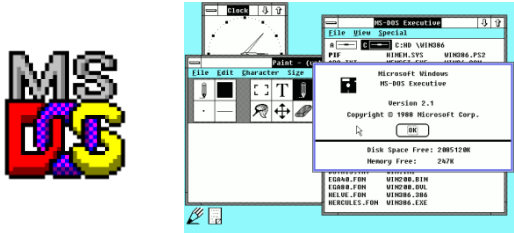
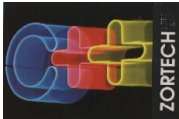
19

2. NewThing **availability** by design

Cfront: First C++ compiler *compiled to C*, usable “anywhere C is”
– including the C optimizer (perf) and system linker (compat).



Zortech C++: First native C++ compiler.
Available on DOS, Windows SDK compatible.



20

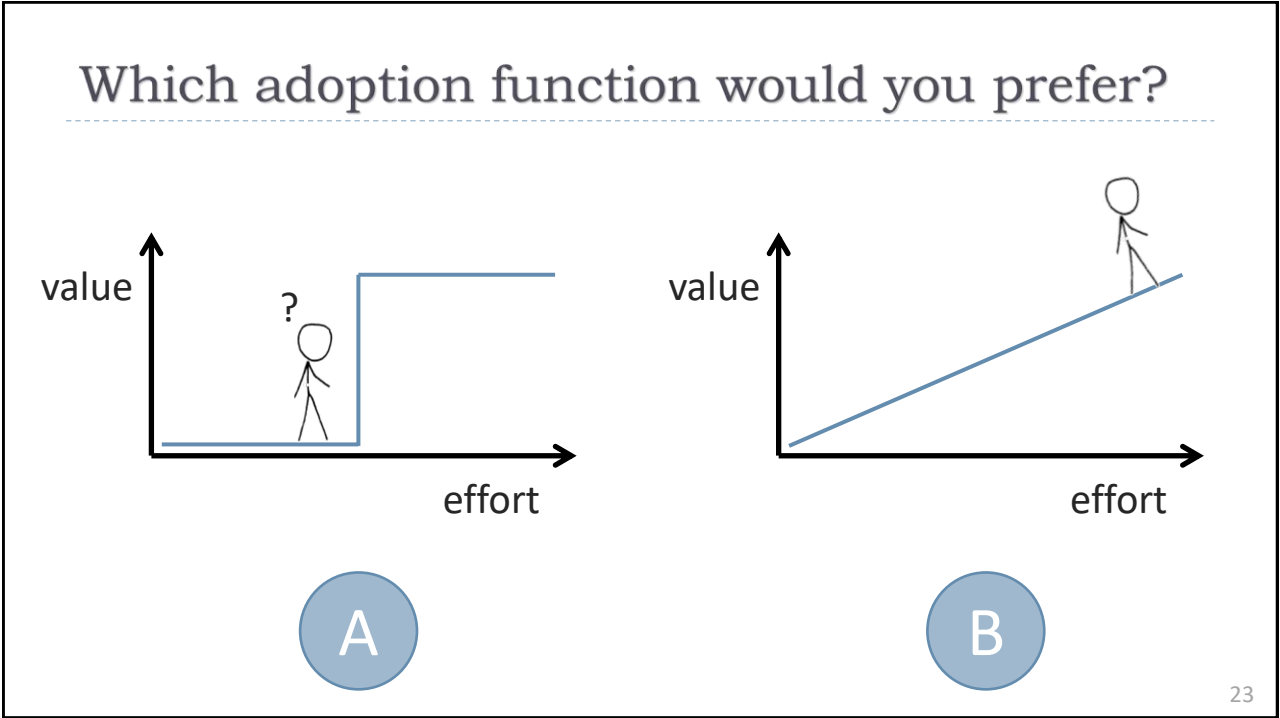
20



21



22



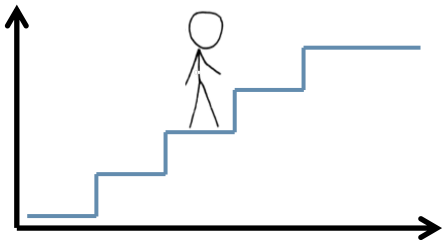
23

23

3. NewThing’s compatibility by design

Basic requirement: **High fidelity interop.**
Min bar: NewThing can seamlessly use OldThing.

Good: “An OldThing project can add NewThing **side by side** and start seeing benefit.”
Ex: “Add NewLang file and see benefit.”



24

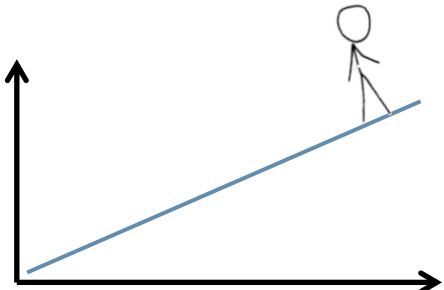
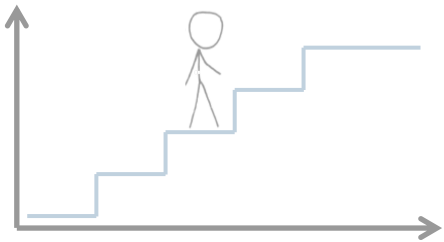
24

3. NewThing’s compatibility by design

Basic requirement: **High fidelity interop.**
Min bar: NewThing can seamlessly use OldThing.

Good: “An OldThing project can add NewThing **side by side** and start seeing benefit.”
Ex: “Add NewLang file and see benefit.”

Grail: “An OldThing project can add NewThing **in one place** and start seeing benefit.”
Ex: “Write 1 line of NewLang and see benefit.”
1980s: Rename `.c` to `.cpp`, add 1 class, benefit.
2010s: Rename `.js` to `.ts`, add 1 class, benefit.



25

25



26

3. NewThing's **compatibility by design**

Lockheed L-1649 Starliner

- ✓ Value: Luxurious, reliable, +10% range
- ✓ Compatible with existing infrastructure



Lockheed Starliner (1956-58)

Boeing 707

- ✓ Value: 2× speed, 2× passengers, quiet
- ✓ **Compatible with existing infrastructure**

Replaced Starliner on all routes by 1962

Retired from passenger service in 2013



Boeing 707 (1958-79)

27

27

3. NewThing's **compatibility by design**

C++: Every C program is a C++ program (still mostly true) + any C++ code can seamlessly call any C + C optimizer+linker.



TypeScript: Every JS program is a TS program + any TS code can seamlessly call any JS code.



Swift: Bidirectional (Swift calls ObjC, ObjC calls Swift), ObjC-friendly object and lifetime models (*ObjC ARC + modules designed for Swift*), automatic bridging header generation, tool support to view ObjC as if written in Swift.



Roslyn next-gen C# compiler: Strict compatibility requirements, adhered to rigorously via compat tests.



28

28

3. NewThing's **compatibility by design**

Compatibility requires strategic up-front design.

Often forgotten until it is too late. Often hard to retrofit.

Improve JS interop #35084



jmesserly opened this issue on Nov 6, 2018 · 8 comments



jmesserly commented on Nov 6, 2018 · edited ▾

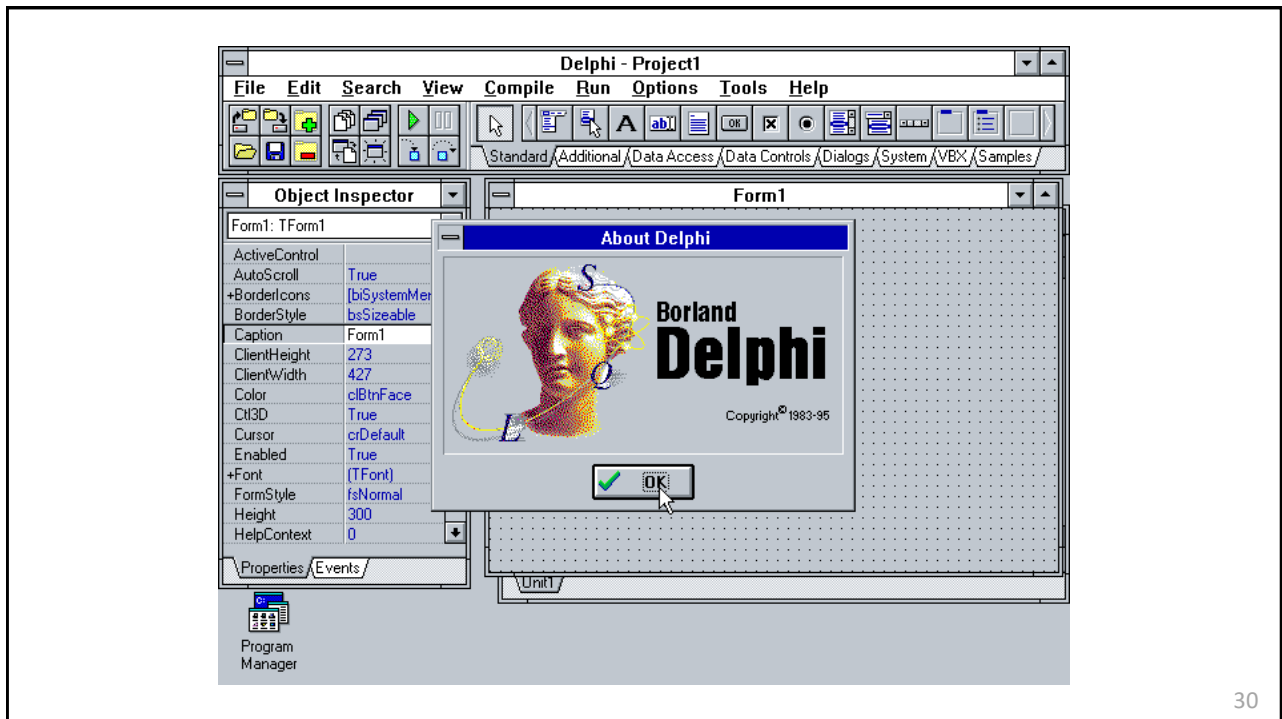
Contributor 😊 ...

We'd like to significantly improve Dart's ability to interoperate with JavaScript libraries and vice versa. This will build on existing capabilities (see [examples of current JS interop](#)).

The new design will take advantage of Dart 2's sound static typing to provide JS interop that is convenient, high performance, and with minimal code size impact.

29

29



30

30

Counterexample: Delphi

Nailed **value** and **availability**

Addressed clear pain: Better RAD, more stable

Ran everywhere VB did: Windows

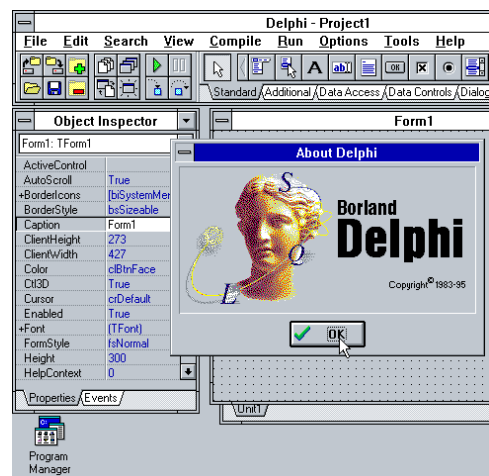
Nailed **compatibility**, initially...

... but couldn't control compatibility

Maintaining VCL in parallel with VBX/OLE/ActiveX tools was problematic

Cost: Prohibitively expensive

Lag: Always behind latest platform features



31

31

Counterexample: Python



2008: Python 3

Source breaking change (can't compile 2 as 3)

	Python 2	Python 3
<code>x = 3/2</code>	<code>x == 2</code>	<code>x == 1.5</code>

Manual migration + tools (2to3, Pylint, Futurize, Modernize, caniusepython3, tox, mypy)

2017: Most Python code still written in “2↔3”

2020: 2.x frozen and unsupported

~12-year transition

vs. 8 years per major version for 1→2→3
(1994→2000→2008)

Python 3 vs Python 2

● Python 3

● Python 2



Although Python 2 is no longer maintained, 10% respondents still actively use it.

Source: JetBrains Python Developers Survey (Oct 2019)

32

32

Counterexample: C99



C99 (1999)

Some additions were controversial and resisted

Fun fact: CPython only allows selected C99 features because of lack of portable compiler support

C11 (2011): Made Complex and VLAs “conditionally supported”
⇒ **optional**, not required for conformance

This third edition cancels and replaces the second edition, ISO/IEC 9899:1999, as corrected by ISO/IEC 9899:1999/Cor 1:2001, ISO/IEC 9899:1999/Cor 2:2004, and ISO/IEC 9899:1999/Cor 3:2007. Major changes from the previous edition include:

— conditional (optional) features (including some that were previously mandatory)

33

33

Counterexample: C++11 string

C++ is highly source & binary compatible (with C & C++prev)
Value (efficient + machine-near) + **bridge** (compatibility)
“Stability is a feature.” – Bjarne Stroustrup

C++11 (2008,11): Banned reference-counted `std::string`
ABI breaking change

GCC 5.1 (2015): First shipped a conforming `std::string`
Then gradually adopted platform by platform (years)

GCC 8 on Red Hat Enterprise Linux 8 (2019):
First turned the conforming string on by default

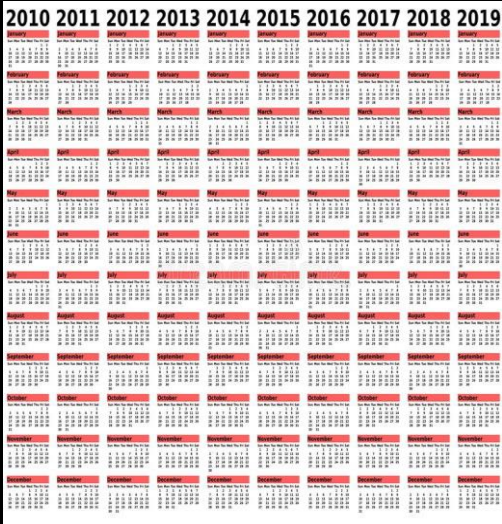


34

34

Quick recap: A “lost decade” pattern

a visual to illustrate
“a decade is a long time”



36

36

Quick recap: A “lost decade” pattern

C99 – ~12 years

Added `_Complex` and VLA in **1999**

Walked them back to “optional” in **2011**

C++11 string – ~11 years

Banned RC for `std::string` in **2008/2010**

Major Linux distro enabled it in **2019**

Python 3 – ~12 years

Shipped 3.0 in **2008**

10% still using 2.x as of early **2020**

If you don’t build a
strong backward
compatibility bridge,
expect to slow your
adoption down by

~10 years

(absent other forces)

37

37



“Every time you take
a sharp turn, some
people fall off”

– Unknown

“Sometimes the
truck falls over”

– Unknown2

38

38

A quick take on two common pitfalls...



39

Pit #1: Annotation requirements

Pitfall: Heavy annotation \Rightarrow **step function**

“Heavy” is often a low number, e.g., ~1 per KLOC

Including viral annotation (aka “color”) \Rightarrow **bifurcated dialect**

Down: “A Red function can only call other Red functions”

Requires bottom-up annotation of the call tree

Lots of work for adopters + canonizes a Red dialect

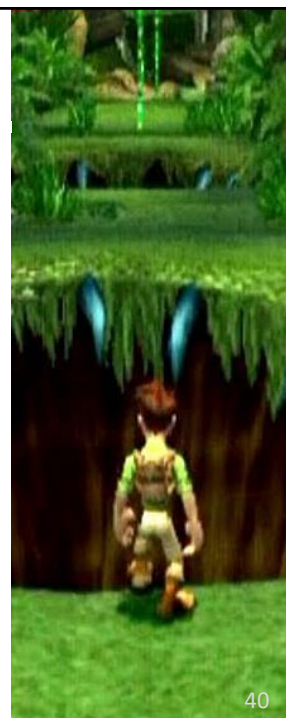
Example: `[[safe]]` functions

Up: “A Red function must be called by Red-aware functions”

Requires here-up annotation of the caller path

Example: **Java checked exceptions**

(common result: “`throws Exception`” opt-out)



40

40

Counterexamples: “Safe C”

Many “Safe C” dialects focused on safe pointers (nullness, lifetime)

2001: **Cyclone** (Morrisett et al.)

2006: v1.0

No longer supported

CYCLONE

2002: **CCured** (Necula et al.)

Issues: **Annotations** + different representations

2005 “Retrofitting”: infer annotations by whole program analysis + extend pointer type system

CCured: Type-Safe
Retrofitting of
Legacy Code

2015: **Checked C** (Tarditi et al.)

“Distinguished by its focus on **backward compatibility, incremental conversion, ...**”

Checked C

41

41

Pit #2: “False friend” incompatibility

Pitfall: “Same code means different things”
⇒ **ambiguous, incompatible dialect**

3/2



Poor choice 1: Allow both in the same source file

Harder to **read** code – need to look at context, lose locality

Harder for **tools** (e.g., refactoring)

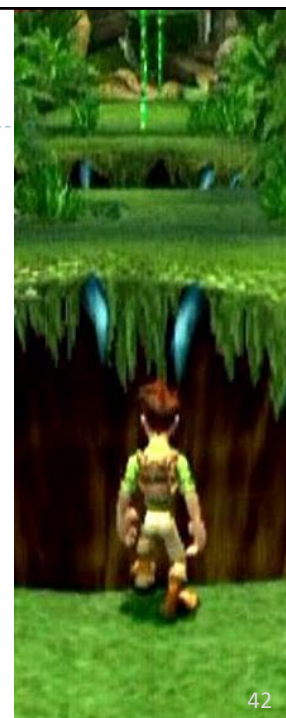
Example: `#pragma __new_syntax, version(2) {...}` block

Poor choice 2: Don’t allow both in the same source file

Can still allow both kinds of files in same project

“Less harder” to read code and create tools

Example: Different source extensions (e.g., `.lang2`)



42

42

So, then...

43

43

“Why will yours succeed, when X, Y, and Z have failed?”

1. Value to address known OldThing pain (and know OldThing’s value).

Real pain needs little explanation. Existing value is often underestimated.

2. Availability wherever OldThing is used.

Explicit design goal from the start, but can grow into it.

3. Compatibility bridge. Seamless backward interop with OldThing.

Explicit design goal from the start. Hard to back into later.

If you don’t, expect to slow your adoption down by ~10 years.

Good: “I can use NewThing **side by side** in an OldThing project.”

Grail: “I can **write 1 line** of NewThing inside OldThing and see benefit.”

44

44

Example: Successor to C++20? (C++23 or NewLang)

1. Value (solving what C++ pain?) + **2. Availability** (wherever C++ is)

“Why will your C++20 successor succeed when *<many>* haven’t?”

Here’s a differentiator that only C++next has tried ... because it’s legit hard ...

3. Compatibility bridge. Seamless backward interop with C++.

Explicit design goal from the start. Hard to back into later.

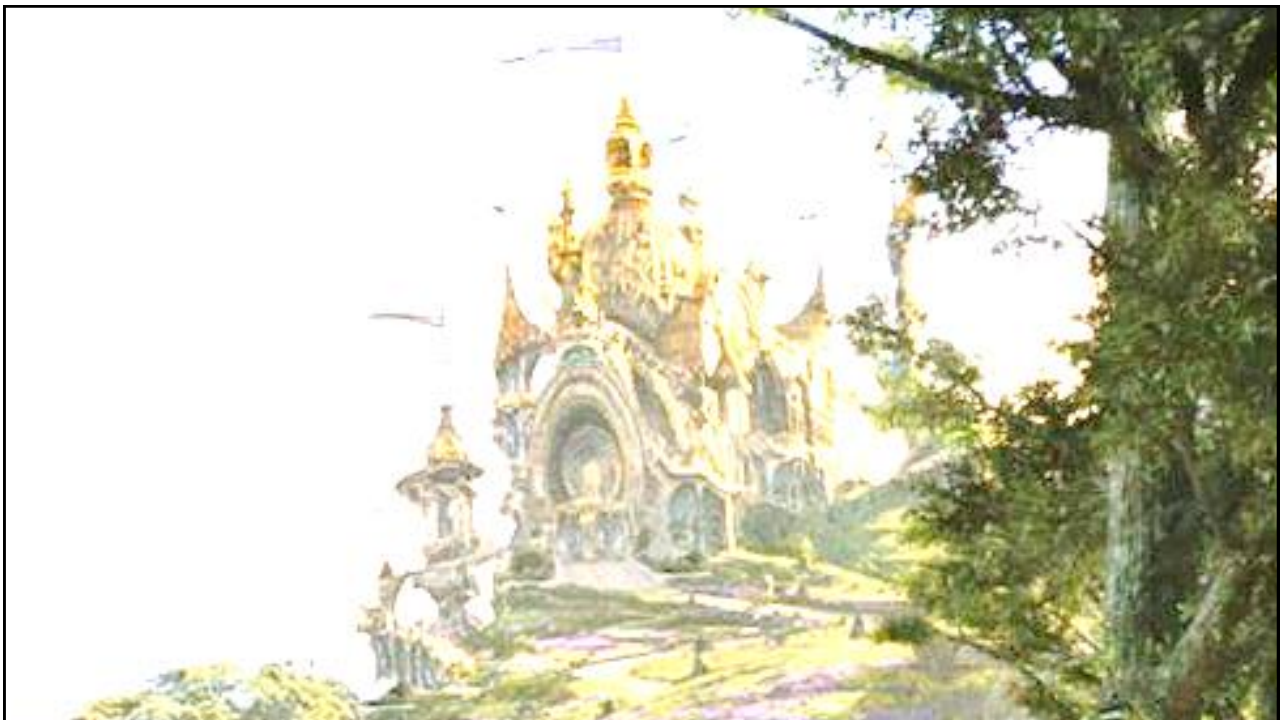
If you don’t, expect to slow your adoption down by ~10 years.

Good: “I can use NewLang **side by side** in a C++ project.”

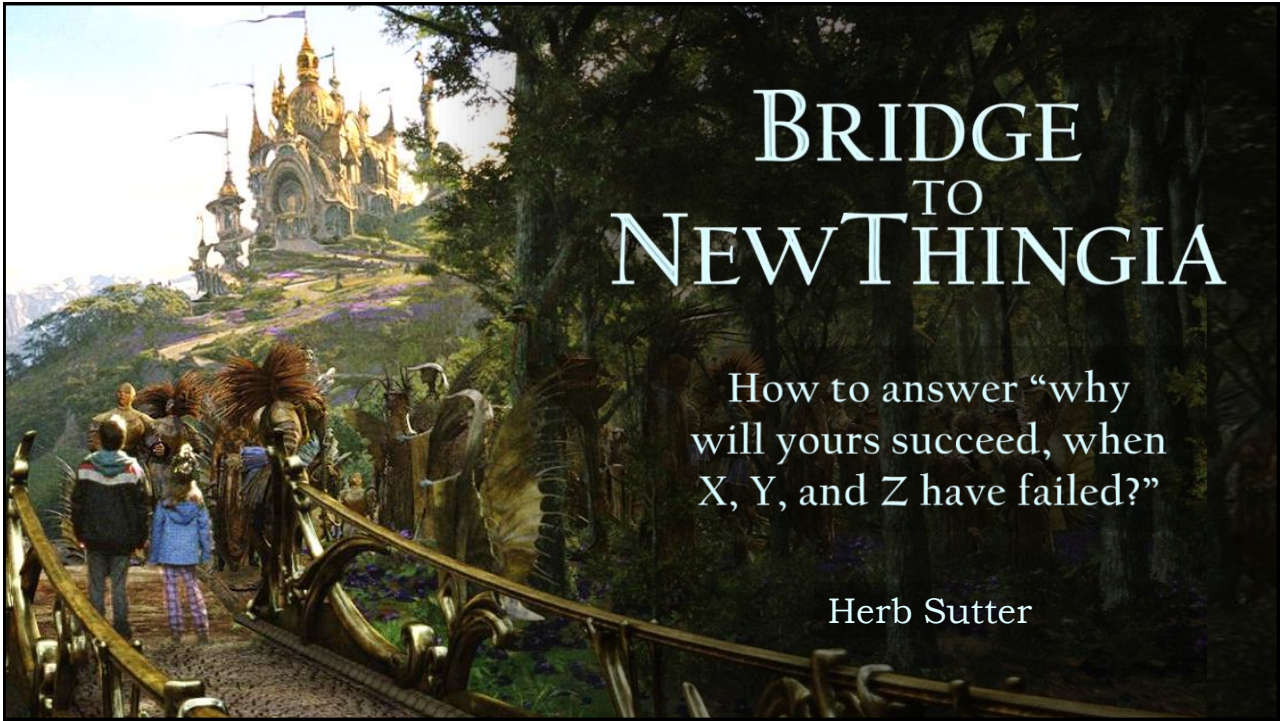
Grail: “I can **write 1 line** in NewLang inside a C++ file and see benefit.”

45

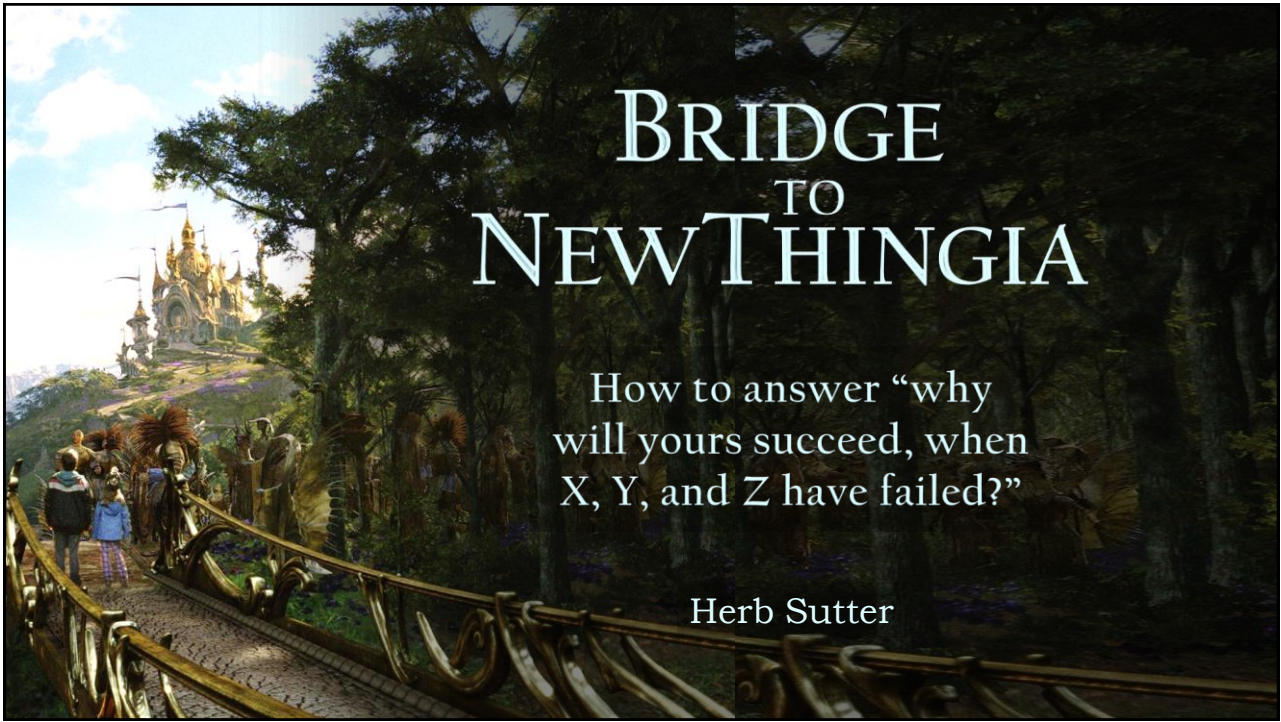
45



46



47



48